

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE Dec 94	3. REPORT TYPE AND DATES COVERED	
4. TITLE AND SUBTITLE A Demonstration of Client-Server Technology Using Remote Procedure Calls with an Applications in File Migration Using Heuristics			5. FUNDING NUMBERS	
6. AUTHOR(S) 2nd LT Eric DeLong			8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/CI/CIA 94-144	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AFIT Students Attending: Arizona State University			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) DEPTMENT OF THE AIR FORCE AFIT/CI 2950 P STREET WRIGHT-PATTERSON AFB OH 45433-7765				
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for Public Release IAW 190-1 Distribution Unlimited MICHAEL M. BRICKER, SMSgt, USAF Chief Administration			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words)				
<div data-bbox="313 1350 751 1749" data-label="Image"> </div> <div data-bbox="1021 1419 1482 1524" data-label="Text"> <p>19950103 053</p> </div> <div data-bbox="594 1656 1053 1791" data-label="Image"> </div>				
14. SUBJECT TERMS			15. NUMBER OF PAGES 39	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT	

A Demonstration of Client-Server Technology

Using Remote Procedure Calls

With an Applications in File Migration

Using Heuristics

by

2LT Eric DeLange
United States Air Force

An Applied Project Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science in Decision and Information Systems

(39 Pages)

Arizona State University

December, 1994

EXECUTIVE SUMMARY

Client/Server technology is one of the fastest growing areas of interest and research in the computer field. Remote Procedure Calling (RPC) is a popular framework for programming in a distributed client/server environment since it facilitates communication between machines operating on different platforms and resembles traditional programming methodologies.

This paper first explores the fundamental concepts behind the implementation of RPC programming. Once a basic explanation of the RPC concept is given, a small example program is illustrated in order to highlight essential elements in any RPC program. A discussion of RPC's applicability in the area of file migration ensues.

Specifically, the utility of RPC in automatically migrating files according to specific rules is examined, such as when a file has been accessed a predetermined number of times from a remote source. An explanation of some of the features of a program developed by the author (in conjunction with others named in the Preface) then follows.

Finally, a way to provide a front end in the HP-UX workspace environment is presented after which the author reviews the lessons learned from and possible amplifications to the project. A bibliography and appendices containing code to the RPC applications are also provided.

Accession For	
ETIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/Availability	
Availability Codes	
Dist	Avail and/or Special
A-1	

Bibliography

- Bloomer, John. Power Programming with RPC. Sebastopol: O'Reilly and Associates, Inc., 1991.
- Carpenter, B. E. and R. Cailliau. "Experience with Remote Procedure Calls in a Real-time Control System." Software--Practice and Experience. Vol. 14. September 1984, pp. 901-07.
- Comer, Douglas E. and David L. Stevens. Internetworking with TCP/IP. Vol 3. Inglewood Cliffs: Prentice Hall, Inc., 1993.
- Curry, David A. Using C on the UNIX System. Sebastopol: O'Reilly and Associates, Inc., 1989.
- Hac, Anna. "A Distributed Algorithm for Performance Improvement Through File Replication, File Migration, and Process Migration." IEEE Transactions on Software Engineering. Vol 15, No 2. November 1989, pp. 1459-1470.
- Hahn, Harley. A Student's Guide to UNIX. New York: McGraw-Hill, 1993.
- Hewlett-Packard Company. HP Visual User Environment 3.0 User's Guide. Hewlett-Packard Company, 1992.
- Kernighan, Brian W. and Dennis M. Ritchie. The C Programming Language. 2nd ed. Murray Hill: AT&T Bell Laboratories, 1988.

Korzeniowski, Paul. "Make Way for Data." Byte. June 1993, pp. 113-115.

Levy, Henry M. and Ewan D. Tempero. "Modules, Objects and Distributed Programming: Issues in RPC and Remote Object Invocation." Software--Practice and Experience. Vol. 21. January 1991, pp. 77-90.

Soto, J. L. Cruz, M. C. Calzada Canalejo and M. Marin Beltran. "Parallelization of Differential Problems by Partitioning Method (Synchronized Algorithm)." Computers and Mathematics with Applications. July 1993, pp. 25-31.

Sun Microsystems. Network Programming Guide. Sun Microsystems, Inc., 1990.

Waite, Mitchell and Stephen Prata. New C Primer Plus. Carmel: Sams Publishing, 1993.

**A Demonstration of Client-Server Technology
Using Remote Procedure Calls With an
Application in File Migration Using Heuristics**

by

Eric DeLange

**An Applied Project Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science in Decision and Information Systems**

Arizona State University

December, 1994

EXECUTIVE SUMMARY

Client/Server technology is one of the fastest growing areas of interest and research in the computer field. Remote Procedure Calling (RPC) is a popular framework for programming in a distributed client/server environment since it facilitates communication between machines operating on different platforms and resembles traditional programming methodologies. This paper first explores the fundamental concepts behind the implementation of RPC programming. Once a basic explanation of the RPC concept is given, a small example program is illustrated in order to highlight essential elements in any RPC program. A discussion of RPC's applicability in the area of file migration ensues. Specifically, the utility of RPC in automatically migrating files according to specific rules is examined, such as when a file has been accessed a predetermined number of times from a remote source. An explanation of some of the features of a program developed by the author (in conjunction with others named in the Preface) then follows. Finally, a way to provide a front end in the HP-UX workspace environment is presented after which the author reviews the lessons learned from and possible amplifications to the project. A bibliography and appendices containing code to the RPC applications are also provided.

TABLE OF CONTENTS

EXECUTIVE SUMMARY.....	2
TABLE OF CONTENTS	3
PREFACE.....	4
INTRODUCTION	6
REMOTE PROCEDURE CALLS.....	9
RPCs vs. LOCAL PROCEDURE CALLS.....	11
A FOUNDATIONAL CLIENT/SERVER DEMONSTRATION.....	13
ADVANCED RPCs.....	16
FILE MIGRATION.....	18
A CASE FOR COMPUTER-INTEGRATED FILE MIGRATION	18
A CASE FOR COMPUTER-INTEGRATED FILE REPLICATION.....	20
THE BENEFITS OF COMPUTER-INTEGRATED FILE MIGRATION.....	21
ANALYSIS OF COMPUTER-INTEGRATED FILE MIGRATION.....	22
THREE APPLICATIONS FOR FILE MIGRATION.....	23
SHARED APPLICATION FEATURES	24
AN APPLICATION USING HEURISTICS	25
RPC FRONT ENDS WITH HP-UX.....	30
THE HP-UX WORKSPACE ENVIRONMENT	30
DEVELOPING A CONTROL.....	31
<i>Building Icons</i>	<i>31</i>
<i>Creating Actions.....</i>	<i>32</i>
<i>Adding a Control to the HP-UX VUE Session</i>	<i>32</i>
RESTARTING THE WORKSPACE	33
CONCLUSIONS.....	35
LESSONS LEARNED	35
FUTURE ENDEAVORS	36
BIBLIOGRAPHY	37
APPENDICES.....	39

PREFACE

This project represents one part of three related and closely coordinated projects. In total, the three projects provide: a) an exposition and programming examples needed to understand the use of Remote Procedure Calling (RPC) for distributed Client/Server processing in networks, b) extensions needed for RPC applications that migrate files among hosts in a network based on several interesting and practical criteria, and c) a discussion pertaining to the development of a Graphical User Interface (GUI) for the on-line demonstration of concepts spanning the collection of projects.

Working as a team, Eric DeLange, Andrew Jank and Andrea Miller jointly participated in developing a tutorial discussion of RPC concepts and programming techniques. After the development of fundamental RPC programs which illustrate these techniques, the team collaborated to develop a foundational set of code pertinent to file migration among networked hosts. Thereafter, each team member individually expanded this code to model a specific RPC application of file migration.

Finally, a jointly-developed GUI to facilitate the convenient on-line execution and demonstration of all applications was added. Source files and a discussion of the GUI are of considerable instructional interest because they present techniques for synthesizing window programming and C programs

that not only make RPC calls but also make system calls to invoke UNIX scripts and utilities.

For the convenience of the reader, each of the related project reports contain the source code and discussion of the applications developed by the companion authors. This project report uniquely contains the discussion and source code developed by the author for file migration using heuristics. Readers who are interested in file migration based on time or immediate file migration are free to reference the reports authored by Andy Jank and Andrea Miller, respectively.

INTRODUCTION

With the vast technological advances in computer capabilities, specifically in the distributed environment, organizations are moving away from centralized systems in order to avail themselves of the various advantages offered by distributed systems. Often, organizations are geographically dispersed which leads them to a natural distribution of computing resources. Furthermore, distribution provides a business with enhanced reliability by allowing data replication across multiple sites. Finally, given the size of organizations in today's business arena, a distributed environment facilitates high transaction rates and also allows for the integration of heterogeneous platforms that often accompany mergers between companies or the addition of new locations.

With the advantages of a distributed environment comes the added complexity of communicating among multiple machines across a network. Since the information is no longer on a single machine, there must be a way to track the identities of the requester and the recipient as well as the location of the information required for the transaction. Client/Server technology is an approach which deals with this complexity. In essence, the requesting computer is viewed as a client which asks for a particular service from another machine (the server). When the client requests a service from the server, control is passed to the server until it has completed processing the request, at

which time the client receives the requested information and regains control. For example, if a client needs the result of a complex mathematical function, but lacks the processing power to compute it within a reasonable amount of time, it can request the services of another processor (server) that has the necessary capabilities to perform the calculation. Once the calculation has been performed, the server passes the answer, and control, back to the awaiting client.

The above example looks very much like a function call in any standard language such as C or FORTRAN and, in fact, it is. The only difference is that the function call is made over a network to a different machine. In truth, making a function call to another machine is aptly accomplished through the implementation of Remote Procedure Calls (RPC). RPC is a popular framework for programming in a distributed client/server environment. It provides a means by which a client can communicate with a server.

This project develops an application using RPC. Specifically, it illustrates the usefulness of RPC and its applicability in the area of file migration. In achieving this end, a small, simple program was developed to provide a basic understanding of RPC. Once a working application in the area of file migration was achieved, each member of the group modified the program to accommodate variations in the file migration

model. Finally, a front-end was added on the Hewlett-Packard UNIX System to make the program more user-friendly.

REMOTE PROCEDURE CALLS

Remote Procedure Calling permits a client to execute procedures on other networked computers. In fact, RPCs serve as the basis for a majority of the distributed system utilities currently in use (i.e. NFS and NIS). A major reason for RPC utilization is the ease with which RPCs can be implemented, when compared to the lower-level network socket interfaces which have been required prior to the advent of RPCs. Moreover, RPCs are perceived as powerful programming tools, especially for users, since their implementation resembles traditional programming methodologies and the network interfaces are provided with increased transparency to the users (Bloomer 1).

RPC has been identified as a type of middleware. Middleware is software that translates communication between different machines or platforms. This type of software protocol is often necessary for communications within a client/server environment. RPC has been deemed one of the two primary types of client/server middleware; the other is message processing (Korzeniowski 114). In order to use RPC, synchronous links between computers must be established, either using datagram or TCP transports. If no transport is available at the time of initiation, the client application will automatically wait for an answer from the server, and eventually "time out" (halt) when no reply is gathered. Message passing differs from RPC protocol in that messaging systems work on the store and forward principle

(Korzeniowski 114). Store and forward systems allow a server to read a computer request message at its own convenience. Since message passing systems do not wait for a response from the server function, these types of systems support asynchronous client/server interaction. However, the synchronous connection supported by RPC communication maintains a higher degree of reliability than message processing. In a message passing system, for example, it is possible that a message may never be received and neither the source, nor the receiver, would be aware of a problem.

RPCs have many other advantages, beyond the simple benefit of enhanced reliability. Some other advantages include the ability of RPCs to run on hosts having different operating systems, the ease of incorporation of RPC into various software products, and the ability to utilize unused CPU time at distant machines. However, RPC implementation has its drawbacks. RPC lacks flexibility and is often difficult to use with many servers. Message passing offers a greater degree of flexibility, along with an easier programming procedure for establishing asynchronous communication between processes in a networked environment. In summary, message passing lacks the reliability and standards that RPC provides and is limited in use (Korzeniowski 115).

RPCs vs. Local Procedure Calls

A remote procedure call appears extremely similar to a local procedure call--as intended. The difference between the two procedures is that in a local procedure call, the client process initiates a procedure in its own address space, whereas with RPC the server and client exist as two separate processes, usually on different machines (Comer 289). It is this separation of processes that allows the server function to reside on a different machine. Nevertheless, it is important to note that RPC can still be utilized when the client and the server execute on the same machine (Comer 306).

During normal implementation of RPC, the client process and the server process communicate to each other via two stubs (Levy 79), namely the client stub and the server stub. A stub is a communications interface that establishes the RPC protocol and determines how each message is constructed by the processes and interchanged between the two. The client process first consults its own stub to locate any remote processes that are required for program operation. Subsequently, the client makes the necessary requests of those processes. Meanwhile, the server (daemon) perpetually listens to the network, through the server stub, for any requests transmitted by clients. More specifically, one daemon, the Inetd, serves as a "grandfather" daemon for all other daemons. Inetd runs perpetually and starts other server daemons upon receipt of requests for server services. The server

fulfills each request in succession, returning to its waiting state after completion of each request (Bloom 2).

At a more basic level, each server process is identified by a port (logical network communication channel) by which it establishes communications for client requests. When a server is initiated on a machine, the computer establishes an address (port) for server communications. This address, which is unique, is registered with the server machine's portmapper (Bloom 11-12).

The portmapper itself provides a crucial network service for all client/server communication. Its job is to keep track of all services that are available on a machine and their port addresses. Whenever a client requests a service from a particular machine, the client petitions the portmapper for the service. If the requested server exists, the portmapper establishes a communication channel between the client and the server. Even when a client and server reside on the same computer, the operation for establishing a link between the client and the server is the same; the network is still involved in the communication. The client still checks with its stub for the server's address, only in this case the address provided by the portmapper would correspond to the same machine. In effect, the request travels across the network only to return to its origin (Sun Microsystems 36).

A Foundational Client/Server Demonstration

To facilitate the comprehension of programs upon approaching the tasks entailing RPC, it is helpful to begin with a straightforward application. For this purpose, a client/server RPC demonstration has been developed wherein a client process passes an integer to a server that increments the integer and returns the updated value to the client. The files necessary to perform this, and any other, RPC application include a protocol definition file, client program, server program, and the stubs and header file generated by the RPC compiler after these are created.

Before programming the client and server processes, it is first necessary to create a protocol definition in the remote procedure call language (RPCL). A protocol definition is a file that describes both the list of data structures that will be passed between the client and server, and the function call required from the client in order to use the server's resources. The initial protocol definition file, *add_it.x*, can be referenced in Appendix A.

The critical elements in the protocol definition include a unique program number and version numbers within each program. In this example, the program is called ADDPROG and is assigned the unique number 0x20000002. It is imperative that this value be unique since it is used by the portmapper to identify the process (from poignant experience, the author can readily attest

to the confusion that results from having multiple programs with the same program number). The version numbers are useful when updates warrant the need for a distinction between the original version, and the subsequent update (Bloomer 43). For example, the `add_it` protocol definition file contains one version with a simple function definition called `ADD_NUM` which receives and returns an integer. Another version could be defined (identified by the number 2) which passes a structure instead of an integer. RPCL is similar to the C language, though this simple example does not make this apparent; however, the protocol definition file for the author's application (Appendix B) demonstrates the similarity between declared constants, structure definitions, and additional functions.

Once the user has created the protocol definition file, a UNIX program, called `RPCGEN`, compiles the definition and produces several files. `RPCGEN` creates both the client and server stubs, as well as a header file that defines the RPC parameters that are included in both the server and client routines. After compilation of the RPC definition file, the next task entails developing the client and server code.

In the `add_it` example, the client code is labeled `add_it.c`. The code is written in C and contains familiar formats like `include` statements, variable declarations, etc. Additionally, the code includes features that are unique to RPC. First, the `rpc/rpc.h` library must be included along with the header file

`add_it.h` which is produced by the compiler `rpcgen`. Next, a special pointer of type `CLIENT` is declared which points to a structure that contains information about the port and socket addresses (Bloomer 7). The value of that pointer is determined by the function `clnt_create` which establishes a connection between the server and client machines. This function requires the name of the host with which to establish a connection (can be the same), program name, version name, and the type of transport protocol (`tcp` or `udp`). If no connection can be established, the function `clnt_pcreateerror` is called to inform the user that no connection could be made to the host (Sun Microsystems 45). Finally, the function that was declared in the protocol definition file (`add_it.x`) is called (the version number is appended to the function name by `rpcgen`). Passed to the function are the integer which will be manipulated and the `CLIENT` pointer which contains the communication information.

The last component of the `add_it` example is the server code, which is also coded in the C language. Again, the `rpc/rpc.h` library is included as well as the header file `add_it.h`. Noticeably different from the client code is the lack of a main declaration. In essence, the server code is simply a function declaration and can be considered as a function within the client code, only residing on a different machine. Furthermore, all communication between the client and the server is accomplished through pointers. Thus, the client passed the pointer to the

integer *num* and the server returns a pointer to the new number which is, ironically, called *oldnum*. The descriptive "Hello People" statement was included to provide optional output to confirm that the server was responding (debugging tool).

Advanced RPCs

Once this simple RPC application that adds two integers over a network was completed, the level of difficulty was increased by working with strings and structures, until the author was versed enough in RPC protocol and application specification to begin work on the file migration application itself. It uses many of the fundamental concepts of the simpler RPC application, as well as additional, more complex concepts. A copy of the code is included in Appendices B through E. Appendix B is the protocol definition file and client code, Appendix C contains the server code, Appendix D contains the protocol definition and server code for the second server (*fts.x* and *fts_svc_proc.c*), and Appendix E has all of the scripts written in UNIX that are used through system calls by the client and server code.

The application of file migration, of course, is not the only area where RPC can be employed effectively. There are many other applications which lend themselves to the advantages of RPC, including using RPC to calculate partial derivatives. In this application, the equations are partitioned and numerically

solved on different computers. The fragmented solutions are then pooled to obtain the final result (Soto 25-27). Another application area encompasses using RPC as a tool for software applications dealing with real-time process control systems that are large and complex (Carpenter 901-902). Undoubtedly, the use of RPC will become more widespread and, as it does, we will see an increase in the number of applications in this area.

FILE MIGRATION

Many organizations have expanded significantly, in a physical and geographical sense, over the past few decades. This growth has been accelerated by the fast-paced nature of the advancing computing environment. Many organizations have noticed that it can be beneficial to modify the location of various frequently accessed company files from their current locations to others. By varying the location of files appropriately, communication costs and file transfer duration times, which typically result from locating and acquiring large files (such as company accounting histories or files containing multiple graphic bitmaps), can be minimized. Companies have often solved these problems by manually modifying the primary location of particular files, or by replicating the files to multiple locations.

A Case for Computer-integrated File Migration

For example, most world-wide corporations maintain personnel files for each and every member of their company. Regardless of each member's current location, it is imperative that the data contained within each file remains current. In order to maintain currency, the file must be updated continually as to reflect each person's current location, position, job status, possible job qualifications, personal preferences, etc. Most companies

maintain each member's personal record at one specific site. The file is usually located closest to where that person normally conducts business (at some home-base location). Whenever their particular personnel file must be updated, their central file must be found and modified. Although this method may seem prudent, as long as the person remains at that specific location for an extended period of time and rarely deviates far from that location on business ventures, the reality of global business dictates that many personnel are frequently dispatched world-wide for extended periods of time at irregular intervals.

If a person is temporarily reassigned from one business site to another, for example, as the schedule of a sales representative could require, maintaining a file at a central location may be considered unwise. It makes sense, then, to have each company member's personal file "follow" them to their current business location, as required by their occupation. Many organizations have implemented this concept by manually moving personnel files from one location to another, as warranted. The process of moving this file is usually performed manually, but why bother when a computer system can perform the necessary file transfers, at close to optimal times, and minimize transferal and access costs concurrently? In most cases, the manual process of moving files is either cost-ineffective, inefficient, or subject to oversight errors.

A case for Computer-integrated File Replication

Related to the prior example, a person may perform certain business functions at two or more distant locations on a frequent and extended basis. In order to minimize communication costs, their personal file could be manually moved between these locations as their business functions require. However, this method may not be the most efficient and cost-effective one for determining the timing of file relocations. Routinely moving a file among two or more locations seems like a senseless task. If the employee regularly returns to a limited set of specific locations, having information specialists repeatedly relocate the member's personal file to each of these locations could result in unnecessary file transfer costs. If this person's file is only needed at each location for read-only applications, a simple file copy at each location would minimize the cost of access times and would require only one file transfer for each location. Any competent computer specialist could determine when a file should be copied to another location, but why employ one if the computer system can perform this function on its own? The system could apply a set of criteria and automatically determine when a file should be copied to another location. The possibility of automatically moving files to the optimal locations leads to the concept of file migration.

The Benefits of Computer-integrated File Migration

The two preceding examples illustrate the advantages of computer-integrated file migration. Although many methods can be employed to determine when a file should be either replicated or moved, there is little evidence to support that the file transfers themselves should be performed manually by the computer system monitors. A computer routine can be developed that determines the optimal distribution of each file and performs the necessary relocation(s) without human involvement.

The concept of file migration is, more or less, an automatic process. In this process, the organization pre-defines a specific set of criteria that must be met before a file is either relocated or replicated to another site. When the pre-specified criteria are met, the awaiting computer functions perform the desired file relocation and replication without any interaction by the computer system monitor. In this scenario, a centralized database would maintain the criteria as well as current information on the location of each file, the type of each file, and the types of accesses available to each file (e.g. if a file is read-only, write-only, or both). If the file location(s) have been modified, only a simple change to the database is required.

Although it may seem expensive to require each file query to consult the central database prior to any, and every, file access, the cost will usually be negligent, especially when compared to normal file transfers. For example, a personnel file

that contains complex images (including dental exam x-rays, photographs, and other bitmaps) can often take hours to transfer error-free over a large distance, while a simple database call to another machine takes much less than a few seconds.

Analysis of Computer-integrated File Migration

So, as far as the added communication overhead is concerned, file migration adds little to the overall cost. Actually, this extra overhead cost is required from all personnel database systems, since any distributed system requires the existence of, and access to, some type of database to determine a particular file's location. The real savings from the file migration concept comes from the efficiency gained by the elimination of human interaction at the file transfer level. The transferal/replication criteria are only specified at a single point, with periodic updates, eliminating the necessity of human decision-makers to determine the location of each file at each point in time. The system itself takes care of the implementation issue.

THREE APPLICATIONS FOR FILE MIGRATION

In order to demonstrate RPC, three different, but related, file migration applications were chosen with regard to unit personnel records. For instance, in the Air Force, currently all records are kept on one database in one location. While this may seem like the simplest way to maintain a database, it might be more cost-effective to distribute the files and move them electronically. This is the basic idea behind the applications. While each option is a slightly different scenario with respect to migrating personnel records, all three applications have an underlying goal as well as a communal set of code.

This shared objective is to create an application that will maintain databases of the actual personnel records along with a database that knows the location of each record. Furthermore, each application will display a set of statistics regarding the location and number of accesses of a particular file from a particular location. This enables an individual to keep track of where the file is used and requested most often. Finally, based on some predetermined set of rules (here is where the different options are derived) the file is transferred to a different location and a message appears telling the user that the file has been transferred to a new location. The transfer is also updated in the database that tracks the locations of the files.

As mentioned before, the file transfer is governed by a set of rules. Here is where each application finds its identity.

The first option deals with transferring a file based on a location that is different from the current location of the file. The second option transfers a file based on the percentage of requests made. The final option transfers a file at regularly scheduled times to given locations. There exist complex algorithms that dictate when a file should be migrated based on a system's resource utilization, file sizes, and the number of write and read accesses to those files (Hac 1459); however, because the emphasis of this paper is on remote procedure calls rather than file migration techniques, these algorithms are beyond the scope of this project.

Shared Application Features

Though each of the three applications address a different decision rule, they all share certain attributes, processes, and implementation techniques. Since the authors developed a generic set of code around which the applications are built, each application shares a number of the same variables. Furthermore, many of the functions within the server code are identical since the same file information is being processed in each application.

Of special interest is the use of the rdbpt server as a client of the fts server. In other words, the same process can function as a server and a client! The server process need only make a request of another server to become a client, too. This feature in the code required extensive debugging for two reasons:

first, the location in the code of the second server call was important to the server's operation and second, the structure of the compilation file was altered significantly in order to integrate the additional stubs into the original configuration.

Finally, each application implements C's capability to make UNIX system calls. This capability allows the integration of UNIX scripts which are triggered by C with a simple system call to the name of the script. In this way, the various tasks of the applications are performed in the most appropriate environment. For example, it is easier to work with files using UNIX rather than C since the code resides on UNIX machines. The UNIX system calls are easily identified throughout the code.

An Application Using Heuristics

Though moving a file to a new location every time that it is requested from that location may be worthwhile and advantageous in some situations, there certainly exist a greater number of situations where doing so every time would amount to an incredible communication cost due to the number of times a certain file may be accessed by a variety of sites. Perhaps, then, the situation may dictate that a file migrate to the location from which it is **most** requested.

In this file migration scenario, established rules are applied that determine when and to where a file is transferred. For example, a rule can be established that every 20 times a file

is accessed, it is migrated to the server which accessed it most out of those 20 times. Moreover, an additional qualifier can be added which determines the percentage of that server's access compared with the other servers; thus, if the percentage is above an established level, the file is moved. Of course, such a rule is overly simplified and the procedures for determining the point at which a file might be transferred would involve complex algorithms considering a number of variables such as file size and system resource utilization. Notice, however, that if the files that are relocated according to rules that satisfy the spirit of this application are base tables or base table fragments, then this application addresses the very important task of dynamically and automatically reconfiguring distributed databases in response to evolving usage.

The program itself can be initiated either by clicking on the appropriate icon which has been created on the upper control bar (reference the ensuing section) or by typing `run_proj2` at the command line. This command starts the two server processes (`rdbpt_svc_proc.c` and `fts_svc_proc.c`) and then waits for the client command which will request those services. By typing `rdbpt hpnotic 11111111` at the command line, for example, the user is invoking the client stub to find the file information on an individual with the identification number 11111111 through the server which resides on the host called `hpnotic`. The call to

the server actually takes place on lines 59 and 60 of the *rdbpt.c* code in Appendix B.

The server is first invoked to find the file in the database (if it exists) and to record the origin of the request. Since the database tracks how often a certain file has been requested from each host, rules can be implemented which move files based on the number of accesses. Lines 73 through 95 of the server code *rdbpt_svc_proc.c* show the example rules used in this application. Once more than five requests on a file have been made, the host which has requested that file the most is found

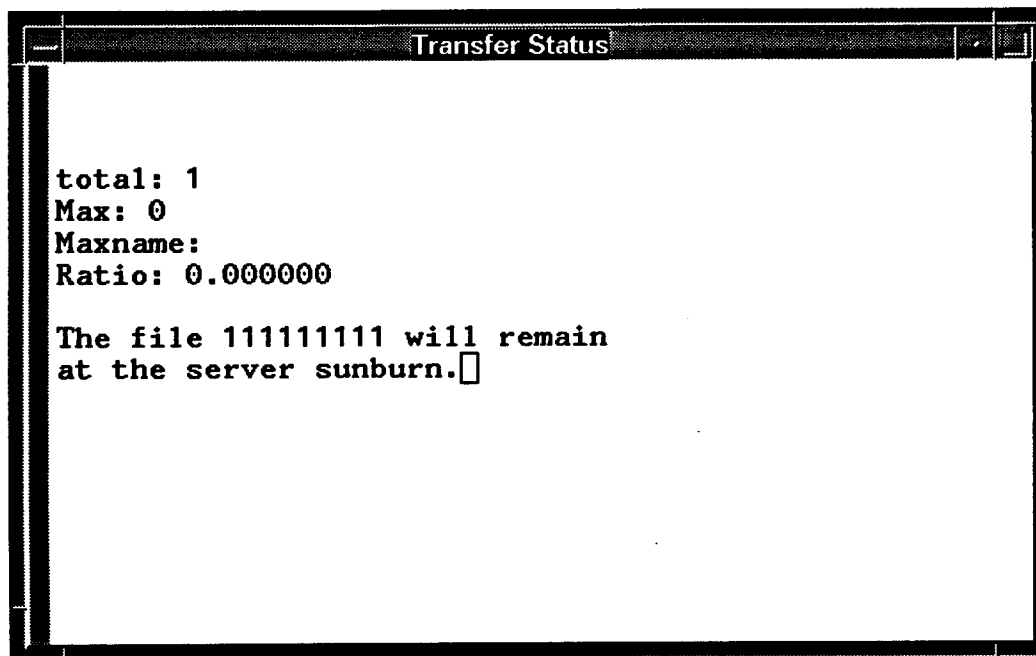
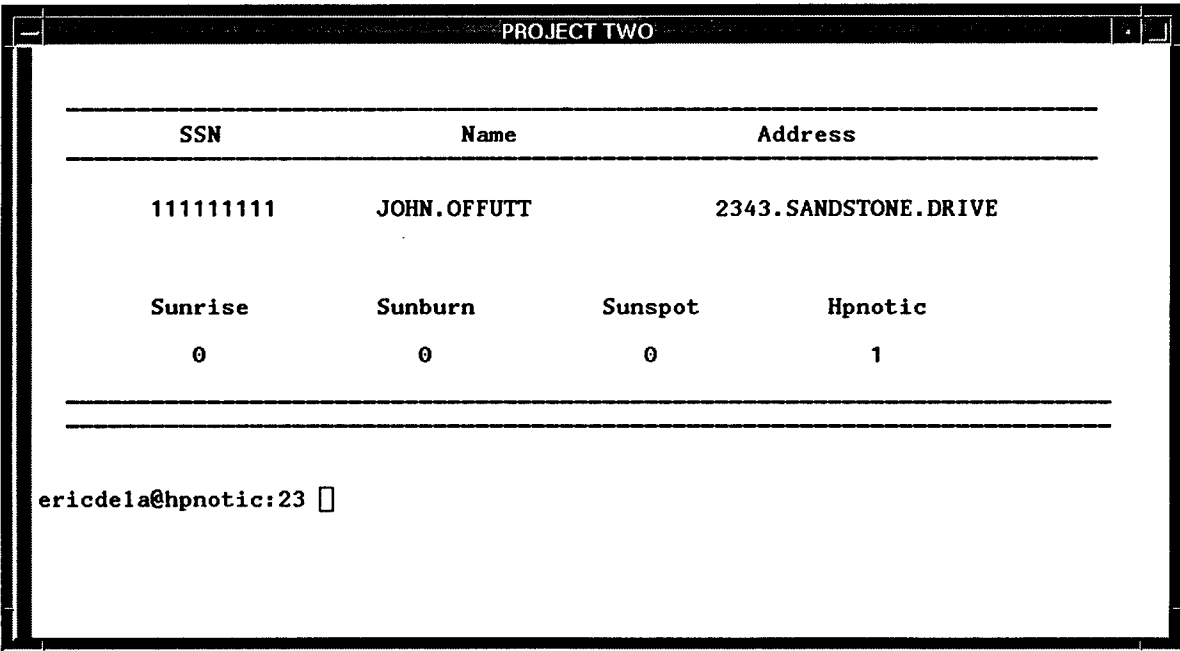


Figure 1

and a ratio is computed. If the ratio exceeds a determined percentage (in this case 0.2), then the file is moved (line 91). Figure 1 shows the output which the Transfer Status window shows. If the total for the file 111111111 were 6, all of which were

requested by the host *hpnotic*, then the maxname would be *hpnotic*, the ratio would equal 1.0, and the message would report that the file 111111111 has been moved from *sunburn* to *hpnotic*.

Regardless of whether or not the file is moved, the user still receives the report illustrated in Figure 2. This window



SSN	Name	Address
111111111	JOHN.OFFUTT	2343.SANDSTONE.DRIVE

Sunrise	Sunburn	Sunspot	Hpnotic
0	0	0	1

ericdela@hpnotic:23

Figure 2

shows the name, address, and the number of times the file has been requested from the four hosts listed. In the example, the file 111111111 has only been requested once from the host *hpnotic*.

Line 223 of the server *rdabt_svc_proc.c* code is notable since that is where the server takes on the additional role of a client. If this application were to physically transfer the

files (whereas it currently only simulates such an action), the server *fts_svc_proc.c* would be the mechanism for the transfer. Lines 218 to 222 ensure that the server exists, and if so, establish the connection. Then, line 223 makes the call to the server (the code for the *fts_svc_proc.c* server is located in Appendix D), which in this case reports whether or not the file has been moved.

RPC FRONT ENDS WITH HP-UX

The front-end display is an important addition to any application, especially for users. Applications that use RPC technology typically produce a large number of complicated, and almost indecipherable, files. Additionally, in order to run any RPC application, the server application must already be waiting for a request at the server end, before the client can be executed. The typical user does not know, or care, about any of the implementation issues. One attractive quality of client/server computing is that the complicated aspects of computer integration and application are transparent to the user. Therefore, it is important that any user interface that utilizes RPC functions facilitates running the program, and all of its additional files, without involving the end-user in the details of its underlying operation.

The HP-UX Workspace Environment

The HP-UX machines provide several different viewing sessions that offer applications and tools that can be accessed by simply clicking a pointing device (ex. a mouse) on an icon. Two of these sessions are HP VUE and HP VUE lite. The basic difference between the two is that the HP VUE lite session has a front panel with different features and different methods for

determining how the view manager interprets the information provided by the user. These panels are typically displayed whenever the user logs on, much how Microsoft Windows provides a set of panels, each of which contains a set of program icons. Users can customize their own panels, but only within the limits set by the System Administrator, who determines what type of options are available to each class of user.

Developing a Control

The application was developed within the HP VUE session. The HP VUE session panel consists of a top and bottom row. In order to modify these rows several changes must be performed: an icon must be constructed, an action that will be associated with that icon must be defined, the action must then be linked to the icon to define a control, and finally the control (the icon with an underlying action) must be added to the front panel by editing the ".vuemrc" file.

Building Icons

Creating an icon is relatively easy for anyone who has a creative mind and likes to draw. The IconEditor is specifically designed for this purpose; it provides the user with a drawing screen and tools to construct various icons. Once the icon is finished, the constructor need only save it for future use. In

order to have the icon associated with an application, an action must be specified and attached to the icon.

Creating Actions

Creating an action is accomplished by simply using the CreateAction icon in the HP-UX General Toolbox. The user inputs the action name, a command line to initiate the application, and a type of window to display the action results within. An action can only be specified once; if future modifications must be made, a new icon-action definition must be specified. Once the icon and action associated with it have been created, the newly defined control can be added to the panel.

Adding a Control to the HP-UX VUE Session

The ".vuemrc" file must be edited to include the new control. The ".vuemrc" file contains all the control definitions, as well as the names for the controls that reside in the top and bottom rows of the panel. These controls can take on different actions, as well as combinations of behaviors including: push button, drop zone, file monitor, client window, and toggle button.

To include a new control to the ".vuemrc" file, several steps must be taken. Initially, the icon controls are placed in the appropriate box and are then defined. The box placement describes in what order the various controls will be displayed.

The word "CONTROL" along with the name for a new control are included among the other control definitions for a desired box (top or bottom) and position within the box. The type of control must also be defined. In the case of the application, a button "TYPE" control is used to initiate the application. For any control addition, the "PUSH_ACTION" for the button must also be specified. The "PUSH_ACTION" can either be an action similar to the one created above, or an executable command. An action similar to the one created above is incorporated into the author's RPC application. Finally, an "IMAGE" must be specified for each control. The image is simply the icon name. Little more is involved with modifying the view configuration of the HP-UX machines to include any, and all, new applications to the window manager.

Restarting the Workspace

In order to incorporate the new controls to the window manager, all one needs to do is restart the workspace manager. The new control definitions will take effect and function in all future uses of the start-up window manager workspace. By using icons and associating them with actions it is possible to execute the RPC application in an environment that not only preserves the benefits of client/server computing, but also retains the

invaluable element of end-user transparency by isolating the user from the underlying complications involved with RPC technology.

CONCLUSIONS

Though the author's experience in the area of RPC is limited, by researching the subject and working on a project that incorporates its capabilities, the usefulness of RPC in communicating in a client/server environment was recognized. Moreover, even though the project was somewhat limited, the complexity involved in implementing client/server technology is readily discerned. Of course, the project was not completed without learning something, nor were the possible applications of RPC in the realm of file migration exhausted.

Lessons Learned

As with any project, there is no substitute for the lessons one learns from experience. This project is no exception. First and foremost, it cannot be stressed enough how important it is that the program number defined in the protocol definition file be unique. If there are two programs with identical values, a clear communication channel cannot be established between a client and **one** server since, according to a client's stub, two server's will exist for the same purpose.

Another important issue in dealing with RPC is passing parameters. It is necessary that all variables be passed as pointers for RPC to work properly. Additionally, the value that is to be returned from the server function must be declared as a static variable.

Future Endeavors

In order to keep the scope of this project within reason, the three file migration applications were approached from a simulation perspective. That is, no physical migration of files actually occurs. When a file is said to have been moved from one host to another, the only thing that has changed is the database file that would track the migration of files. Obviously, this leaves open the possibility of expanding the project by physically migrating the files. Furthermore, additional applications can be developed such as maintaining duplicated files at separate sites (read-write and read-read access are important in this area). Along with this, more research can be done in adequate rules that define at what point a file is to be moved or duplicated. The author recognizes that this does not comprise an exhaustive list of supplementary application areas to the project and the reader is free to explore their own possibilities.

BIBLIOGRAPHY

- Bloomer, John. Power Programming with RPC. Sebastopol: O'Reilly and Associates, Inc., 1991.
- Carpenter, B. E. and R. Cailliau. "Experience with Remote Procedure Calls in a Real-time Control System." Software--Practice and Experience. Vol. 14. September 1984, pp. 901-07.
- Comer, Douglas E. and David L. Stevens. Internetworking with TCP/IP. Vol 3. Inglewood Cliffs: Prentice Hall, Inc., 1993.
- Curry, David A. Using C on the UNIX System. Sebastopol: O'Reilly and Associates, Inc., 1989.
- Hac, Anna. "A Distributed Algorithm for Performance Improvement Through File Replication, File Migration, and Process Migration." IEEE Transactions on Software Engineering. Vol 15, No 2. November 1989, pp. 1459-1470.
- Hahn, Harley. A Student's Guide to UNIX. New York: McGraw-Hill, 1993.
- Hewlett-Packard Company. HP Visual User Environment 3.0 User's Guide. Hewlett-Packard Company, 1992.
- Kernighan, Brian W. and Dennis M. Ritchie. The C Programming Language. 2nd ed. Murray Hill: AT&T Bell Laboratories, 1988.
- Korzeniowski, Paul. "Make Way for Data." Byte. June 1993, pp. 113-115.
- Levy, Henry M. and Ewan D. Tempero. "Modules, Objects and Distributed Programming: Issues in RPC and Remote Object Invocation." Software--Practice and Experience. Vol. 21. January 1991, pp. 77-90.
- Soto, J. L. Cruz, M. C. Calzada Canalejo and M. Marin Beltran. "Parallelization of Differential Problems by Partitioning Method (Synchronized Algorithm)." Computers and Mathematics with Applications. July 1993, pp. 25-31.
-

Sun Microsystems. Network Programming Guide. Sun Microsystems, Inc., 1990.

Waite, Mitchell and Stephen Prata. New C Primer Plus. Carmel: Sams Publishing, 1993.

Appendix

A

```
1  /* ADD_IT.X */
2  /* THIS FILE IS USED BY RPCGEN TO PRODUCE THE HEADER FILE ADD_IT.H */

3  program ADDPROG {
4      version ADDVERS {
5          int ADD_NUM(int) = 1;
6      } = 1;
7  } = 0x20009300;
```

```
1  /*      ADD_IT.C      */
2  /* THIS IS THE CLIENT PROCEDURE */

3  #include <stdio.h>
4  #include <ctype.h>
5  #include <rpc /rpc.h>
6  #include "add_it.h"

7  main()
8  {
9      CLIENT *c1;
10     int num;
11     char *hostname[20];

12     printf("\nPlease enter the remote host name: ");
13     gets(hostname);

14     printf("\n\nPlease enter the number to increment: ");
15     scanf("%d", &num);

16     /* THIS IS THE CLIENT HANDLE THAT ESTABLISHES THE CONNECTION */
17     /* BETWEEN THE CLIENT AND THE SERVER */

18     if (!(c1 = clnt_create(hostname, ADDPROG, ADDVERS, "tcp")))
19     {
20         clnt_pcreateerror(hostname);
21         exit(1);
22     }

23                                     /* HERE IS THE PROCEDURE CALL */

24     printf("\n\nThe new number is %d\n", *(add_num_1(&num, c1)));
25 }
```

```
1  /* ADD_IT_SVC_PROC.C */
2  /* THIS IS THE SERVICE PROCEDURE */

3  #include <stdio.h>
4  #include <string.h>
5  #include <rpc /rpc.h>
6  #include "add_it.h"

7  int *add_num_1(oldnum)
8  int *oldnum;

9  /* THIS FUNCTION ADDS 21 TO THE INPUT NUMBER AND THEN RETURNS THE NUMBER */
10 {
11     printf("\nHello People");
12     *(oldnum) += 21;
13     return oldnum;
14 }
```

Appendix

B


```

1      /*                      Project 2
2      /*
3      /*
4      /*                      RDBPTX
5      /*
6      /*                      This program is used by RPCGEN to produce the header file
7      /*                      for the RDBPT RPC.
8      /*

9      const MAX_REC_LEN = 255;          /* max personal record length
10     const SSN_SIZE = 9;                /* size of Social Security Number
11     const NAME_SIZE = 80;              /* size of person's name
12     const ADDR_SIZE = 80;              /* size of person's address

13     const HOST_SIZE = 255;             /* size of host computer

14     /*                      Defines the structure of each personal record:
15     /*                      SSN, name, and address
16     /*

17     struct pers_rec
18     {
19         string      ssn<SSN_SIZE>;
20         string      name<NAME_SIZE>;
21         string      address<ADDR_SIZE>;
22     };

23     /*                      Defines the structure of the database record for each file:
24     /*                      SSN (filename), current location, and four counters to measure
25     /*                      file accesses from each server.
26     /*

27     struct dbase_rec
28     {
29         string      ssn<SSN_SIZE>;
30         string      loc<HOST_SIZE>;
31         int         sunrise;
32         int         sunburn;
33         int         sunspot;
34         int         hpnotic;
35     };

36     /*                      Defines the structure of an information record about a specific
37     /*                      file, including the SSN (filename), the local host from the
38     /*                      file has been requested, and where the file is actually located.
39     /*

40     struct inputrec
41     {
42         string      ssn<SSN_SIZE>;
43         string      lc_host<HOST_SIZE>;
44         string      h_name<HOST_SIZE>;
45     };
46

```

```
47  program RDBPT_PROG
48  {
49      version RDBPT_VERS
50      {
51          dbase_rec SSN_KEY(inputrec) = 1;
52          pers_rec GET_REC(inputrec) = 2;
53      } = 1;
54  } = 0x20009303;
```

```

1  /* */
2  /* *** project 2 *** */
3  /* */
4  /* RDBPT.C */
5  /* */
6  /* This is the client code for the RDBPT service. The */
7  /* client receives as input the hostname and the filename */
8  /* for access. The code translates these inputs into the */
9  /* proper form, contacts the RDBPT server to find the */
10 /* actual location of the file, then retrieves and prints */
11 /* the file for the user. */
12 /* */

13 #include <stdio.h>
14 #include <string.h>
15 #include <rpc /rpc.h>
16 #include "rdbpt.h"
17 #include <stdlib.h>

18 main (argc, argv)
19     int      argc;
20     char     *argv[];
21 {
22     CLIENT    *cl;           /* Client handle */
23     dbase_rec *d_record;     /* Retrieved dbase record */
24     pers_rec  *p_record;     /* Retrieved personal record */
25     inputrec  inrec;         /* Information Record (Argvs) */
26     FILE      *c_fp = NULL;  /* FP for hostname conversion */
27     FILE      *ssn_p = NULL; /* FP for SSN conversion */
28     char       HSTNAME[HOST_SIZE]; /* temp var for hostname */

29     /* Assigne filename and hostname to variables for use */

30     inrec.ssn = argv[2];
31     inrec.h_name = argv[1];

32     /* Convert hostname to proper representation */

33     system ("find_hname");
34     c_fp = fopen("hname.txt","r");
35     fscanf (c_fp, "%s", HSTNAME);
36     fclose (c_fp);
37     inrec.lc_host = HSTNAME;

38     /* Convert SSN filename to proper representation */

39     ssn_p = fopen("ssn.txt", "w");
40     fprintf(ssn_p, "%s", inrec.ssn);
41     fclose(ssn_p);
42     system("ssn_ident");

43     p_record = (char *) malloc(sizeof(pers_rec));
44     d_record = (char *) malloc(sizeof(dbase_rec));

```

```

45      /*                      Give error usage message                      */

46      if ((argc != 3))
47      {
48          fprintf(stderr, "\nUsage: %s local.server SSN\n", argv[0]);
49          exit(1);
50      }

51      /*                      Establish connection to the server (RDBPT process)                      */

52      if (!(cl = clnt_create(argv[1], RDBPT_PROG,
53                          RDBPT_VERS, "tcp")))
54      {
55          clnt_pcreateerror(argv[1]);
56          exit(1);
57      }

58      /*                      Retrieve file location and access information                      */

59      d_record = ssn_key_1 (&inrec, cl);
60      p_record = get_rec_1 (&inrec, cl);

61      /*                      Print out the contents of the file, the current file location
62      /*                      and the current file access statistics.                      */

63      system ("clear");

64      printf ("\n\n _____");
65      printf ("_____ \n");
66      printf ("%t  SSN\t      Name\t      Address\n");
67      printf (" _____");
68      printf ("_____ \n");

69      printf ("\n\t%s\t%s\t\t\t\t\t", p_record->ssn, p_record->name,
70              p_record->address);

71      printf ("\tSunrise\t\tSunburn\t\tSunspot\t\tHpnotic\n");
72      printf ("\t\t %d\t\t %d\t\t %d\t\t %d\n",
73              d_record->sunrise, d_record->sunburn, d_record->sunspot,
74              d_record->hpnotic);

75      printf (" _____");
76      printf ("_____ \n");
77      printf (" _____");
78      printf ("_____ \n\n");
79  }

```

Appendix

C

```

1  /*          *** project 2 ***          */
2  /*          */
3  /*          */
4  /*          RDBPT_SVC_PROC.C          */
5  /*          */
6  /*          This is the program that provides services to the */
7  /*          RDBPT clients. The server has two main server functions: */
8  /*          one that contacts the central database and retrieves */
9  /*          statistical and location information about the desired */
10 /*          file, and another that actually retrieves the contents of */
11 /*          the personal file.          */
12 /*          */

13 #include <stdio.h>
14 #include <string.h>
15 #include <rpc/rpc.h>
16 #include "rdbpt.h"
17 #include "fts.h"
18 #include <ctype.h>

19 FILE          *fp1 = NULL;          /* FP for accessing central database */
20 FILE          *fp2 = NULL;          /* FP for retrieving personnel record */
21 FILE          *ssn_p = NULL;        /* FP for opening temporary files */
22 static pers_rec *p_rec = NULL;      /* pointer for personnel record */
23 static dbase_rec *d_rec = NULL;     /* pointer for database record */
24 char          *maxname[15];         /* tracks host with most requests */
25 char          *temp_loc[15];        /* temp variable for host info */

26 /*          READ_DBASE          */
27 /*          */
28 /*          This function allocates internal memory and reads the central */
29 /*          database, retrieving the statistics and host information about */
30 /*          the desired personnel file.          */

31 int read_dbase(inrec)
32     inputrec          *inrec;
33 {
34     if (!d_rec)
35     {
36         d_rec = (dbase_rec *) malloc(sizeof(dbase_rec));
37         d_rec->ssn = (char *) malloc(sizeof(SSN_SIZE));
38         d_rec->loc = (char *) malloc(sizeof(HOST_SIZE));
39         d_rec->sunrise = (int) malloc(sizeof(int));
40         d_rec->sunburn = (int) malloc(sizeof(int));
41         d_rec->sunspot = (int) malloc(sizeof(int));
42         d_rec->hpnotic = (int) malloc(sizeof(int));
43     }

44     if (fscanf(fp1, "%s %s %d %d %d %d",
45         d_rec->ssn, d_rec->loc,
46         &d_rec->sunrise, &d_rec->sunburn, &d_rec->sunspot,
47         &d_rec->hpnotic) != 6)
48         return (0);

49     return (1);
50 }

```

```

51      /*                                     WRITE_DBASE                                *
52      /*                                     *                                           *
53      /*                                     This function modifies the statistic counter and file location *
54      /*                                     information and updates the applicable central database record */ *
55      /*                                     *                                           *

56      int write_dbase(inrec)
57          inputrec          *inrec;
58      {
59          int               total, maximum;
60          float             ratio;

61      /*                                     increment the appropriate counter for the host which *
62      /*                                     is requesting the file information *
63      /*                                     *

64          if (strcmp(inrec->lc_host, "sunrise") == 0)
65              d_rec->sunrise += 1;
66          if (strcmp(inrec->lc_host, "sunburn") == 0)
67              d_rec->sunburn += 1;
68          if (strcmp(inrec->lc_host, "sunspot") == 0)
69              d_rec->sunspot += 1;
70          if (strcmp(inrec->lc_host, "hpnotic") == 0)
71              d_rec->hpnotic += 1;

72      /*                                     total all requests from all the hosts *

73          total = (d_rec->sunrise + d_rec->sunburn + d_rec->sunspot +
74                  d_rec->hpnotic);

75      /*                                     Set and evaluate criteria for moving a file based *
76      /*                                     on some percentage of total accesses *
77      /*                                     *

78          system("clear");
79          printf ("\n\n\ttotal: %d", total);
80          if (total > 5)
81              maximum = max();
82          printf("\nMax: %d", maximum);
83          printf("\nMaxname: %s", maxname);
84          ratio = (float) maximum /total;
85          printf ("\nRatio: %f\n",ratio);
86
87
88          strcpy(temp_loc, d_rec->loc);

89          if (ratio > 0.2)
90          {
91              strcpy (d_rec->loc, maxname);
92              d_rec->sunrise = 0;
93              d_rec->sunburn = 0;
94              d_rec->sunspot = 0;
95              d_rec->hpnotic = 0;
96          }

```

```

97      /*                      Write new database changes to disk files                      */

98      fprintf(ssn_p, "%s %s %d %d %d %d\n",
99                d_rec->ssn, d_rec->loc,
100               d_rec->sunrise, d_rec->sunburn, d_rec->sunspot,
101               d_rec->hpnotic);

102
103      return (1);
104  }

105      /*                      INT_MAX                      */
106      /*
107      /*                      This function determines the host which requests a file the
108      /*                      maximum number of times and also tracks the number of times
109      /*                      which it was accessed.
110      /*

111  int max()
112  {
113      int maximum;

114      maximum = d_rec->sunrise;
115      strcpy(maxname, "sunrise");

116      if (d_rec->sunburn > maximum)
117      {
118          maximum = d_rec->sunburn;
119          strcpy(maxname, "sunburn");
120      }

121      if (d_rec->sunspot > maximum)
122      {
123          maximum = d_rec->sunspot;
124          strcpy(maxname, "sunspot");
125      }

126      if (d_rec->hpnotic > maximum)
127      {
128          maximum = d_rec->hpnotic;
129          strcpy(maxname, "hpnotic");
130      }

131      return maximum;

132  }

133      /*                      READ_PERS_REC                      */
134      /*
135      /*                      This function reads the desired personnel record into the
136      /*                      structure p_rec.
137      /*

138  int read_pers_rec()
139  {
140      if (!p_rec)
141      {
142          p_rec = (pers_rec *) malloc(sizeof(pers_rec));
143          p_rec->ssn = (char *) malloc(sizeof(SSN_SIZE));

```



```

144         p_rec->name = (char *) malloc(sizeof(NAME_SIZE));
145         p_rec->address = (char *) malloc(sizeof(ADDR_SIZE));
146     }
147     if (fscanf(fp2, "%s %s %s",
148         p_rec->ssn, p_rec->name, p_rec->address) != 3)
149         return (0);
150     return (1);
151 }

```

```

152     /*                                     SSN_KEY_1
153     /*
154     /*                                     This function is called remotely by an establised client.
155     /*                                     The client sends the local host location and personnel filename */
156     /*                                     and this server returns the actual location of the personnel
157     /*                                     file, as well as the accumulated statistical information about
158     /*                                     that particlar file.
159     /*

```

```

160     dbase_rec *ssn_key_1(input_rec)
161         inputrec          *input_rec;
162     {
163         char *DFILE;
164
165         DFILE = (char *) calloc(MAX_REC_LEN+1, sizeof(char));
166         strcpy(DFILE, "filedb.txt", MAX_REC_LEN);
167
168         if (!(fp1 = fopen (DFILE, "r+")))
169             return ((dbase_rec *) NULL);
170
171         while (read_dbase(input_rec))
172             if (!strcmp(d_rec->ssn, input_rec->ssn))
173                 break;
174
175         ssn_p = fopen("tempdb2", "w");
176
177         if feof (fp1)
178         {
179             fclose (fp1);
180             return ((dbase_rec *) NULL);
181         }
182
183         write_dbase(input_rec);
184         fclose (fp1);
185         fclose(ssn_p);
186         system("/users/andyjank/project2/changefile");
187
188         return ((dbase_rec *) d_rec);
189     }

```

```

184     /*                                     GET_REC_1
185     /*
186     /*                                     This function is called remotely by an establised client.
187     /*                                     The client sends the host location and personnel filename, and
188     /*                                     this server returns the personnel file.
189     /*

```

```

190     pers_rec *get_rec_1(input_rec)

```

```

191         inputrec      *input_rec;

192     {
193         CLIENT          *c2;
194         info_rec        info;
195         char             *number[20];
196         char             *PFILE;
197         pers_rec         *error;

198         PFILE = (char *) calloc(MAX_REC_LEN+1, sizeof(char));
199         strcpy(PFILE, "D.",MAX_REC_LEN);

200         strcat(PFILE, input_rec->:ssn);

201         if (!(fp2 = fopen (PFILE, "r")))
202             return ((pers_rec *) NULL);

203         while (read_pers_rec())
204             if (!(strcmp(p_rec->:ssn, input_rec->:ssn)))
205                 break;

206         if feof (fp2)
207         {
208             fclose (fp2);
209             return ((pers_rec *) NULL);
210         }

211         fclose (fp2);

212         /*                  This code section contacts the file transfer server, which          *
213         /*                  would actually perform the file transfer in a fully implemented      *
214         /*                  application.                                                         *

215         info.oldloc = temp_loc;
216         info.newloc = d_rec->loc;
217         info.filename = d_rec->:ssn;

218         if (!(c2 = clnt_create("hpnotic",FTSPROG, FTSVERS, "tcp")))
219         {
220             clnt_pcreateerror("hpnotic");
221             exit(1);
222         }

223         trans_1(&info, c2);

224         return ((pers_rec *) p_rec);

225     }

```

Appendix

D

```
1  /*          *** project 2 ***
2  /*
3  /*          FTS.X
4  /*
5  /*          This file is compiled by RPCGgen to produce the
6  /*          header file for our file transfer server.

7  const HOST_SIZE = 255;
8  const F_NAME_SIZE = 255;

9          /* Structure that contains the original location of the
10         /* file, the filename, and the desired new file locaiton* /

11  struct info_rec
12  {
13         string      oldloc<HOST_SIZE>;
14         string      newloc<HOST_SIZE>;
15         string      filename<F_NAME_SIZE>;
16  };

17  program FTSPROG
18  {
19         version FTSVERS
20         {
21                 int TRANS(info_rec) = 1;
22         } = 1;
23  } = 0x20009304;
```

```

1  /*          *** project 2 ***
2  /*
3  /*          FTS_SVC_PROC.C
4  /*
5  /*          This file is compiled by RPCgen to produce the
6  /*          file transfer server executable. The executable itself
7  /*          does not actually perform the file transfer to the new
8  /*          server; it only produces a message informing the user
9  /*          that this is where the actual transfer would take place.
10 /*

11 #include <stdio.h>
12 #include <string.h>
13 #include <rpc/rpc.h>
14 #include "fts.h"

15 int *trans_1(h_info)
16 info_rec          *h_info;
17 {
18     static int num;
19     num = 1;

20     if (strcmp(h_info->oldloc,h_info->newloc) == 0)
21         printf("\nThe file %s will remain \nat the server %s.",
22             h_info->filename,h_info->oldloc);
23     else
24         printf("\nThe file %s has been \nmoved from %s to %s.",
25             h_info->filename, h_info->oldloc, h_info->newloc);

26     return (&num);
27 }

```

Appendix

E

```
1  #! /bin /csh -f

2  #                                RUN_IT:                                #
3  #                                #                                #
4  #      This UNIX script starts the two servers and positions          #
5  #      their corresponding windows appropriately.                      #

6  cd /users /andyjank /project2
7  /users /andyjank /project2 /clean
8  /users /andyjank /project2 /makefile

9  xterm -title "Transfer Status" -geometry 53x18+5+5 -e /users /andyjank /project2 /rdbpt_svc&
10 /users /andyjank /project2 /fts_svc&
11 xterm -title "PROJECT TWO" -geometry 80x24+200+400
```

```
1  ##
2  ##                                CLEAN
3  ##
4  ##      This file removes all previously made rpc files.
5  ##

6  kill_fts
7  rm rdbpt_xdr.*
8  rm rdbpt_svc_proc.o
9  rm rdbpt_svc
10 rm rdbpt.o
11 rm rdbpt_clnt.*
12 rm rdbpt_svc.o
13 rm rdbpt.h
14 rm rdbpt_svc.c
15 rm rdbpt
16 rm fts_svc_proc.o
17 rm fts_svc
18 rm fts.h
19 rm fts_svc.*
20 rm fts_clnt.*
21 rm fts_xdr.*
22 rm D.*
23 rm core
24 rm tempdb tempdb2 ssn.txt hname.txt
```



```
1  #! /bin /csh -f

2  ps -e | grep fts_svc | cut -c2-6 >! kill_temp
3  kill `cat kill_temp`
4  rm kill_temp

5  ps -e | grep rdbpt_svc | cut -c2-6 >! kill_temp
6  kill `cat kill_temp`
7  rm kill_temp
```

```
1  #! /bin /csh

2  #                                MAKEFILE:                                #
3  #                                #                                        #
4  #                                This file compiles the rdbpt RPC and forms the needed #
5  #                                object files.                                #

6  makedata
7  rpcgen rdbpt.x
8  rpcgen fts.x

9  cc -c -o rdbpt.o rdbpt.c
10 cc -c rdbpt_clnt.c
11 cc -c rdbpt_xdr.c
12 cc -o rdbpt rdbpt.o rdbpt_clnt.o rdbpt_xdr.o

13 cc -c -o rdbpt_svc_proc.o rdbpt_svc_proc.c
14 cc -c rdbpt_svc.c

15 cc -c fts_xdr.c
16 cc -c -o fts_svc_proc.o fts_svc_proc.c
17 cc -c fts_svc.c

18 cc -o fts_svc fts_svc_proc.o fts_svc.o fts_xdr.o
19 cc -o rdbpt_svc rdbpt_svc_proc.o rdbpt_svc.o rdbpt_xdr.o fts_svc_proc.o
```

find_hname

find_hname

```
1  #!/bin /csh -f

2  #                                FIND_HNAME:                #
3  #                                #                            #
4  #                                This UNIX script finds the  #
   #                                hostname of the client.      #

5  echo `hostname` >! hname.txt
```

```
1  #! /bin /csh -f

2  #                      SSN_IDENT:                #
3  #                                                         #
4  #      This UNIX script takes the input SSN and creates a file #
5  #      that includes all non-matches of that SSN.           #

6  grep -v `cat ssn.txt` /users /andyjank /project2 /filedb.txt > tempdb
```

1 #! /bin /csh -f

2 # CHANGEFILE:
3 #
4 # This UNIX script updates the database file.
5 #

#

6 cat tempdb2 >> tempdb
7 cat tempdb >! filedb.txt